National College of Ireland

Configuration Manual

MSc Research Project
Data Analytics

Emmanuella Anita Odiaka
Student ID: X23282819

School of Computing
National College of Ireland

Supervisor: Abid Yaqoob

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name:	Emmanuella Anita Odiaka
Student ID:	X23282819
Programme:	MSc in Data Analytics **Year:** 2024-2025
Module:	MSc Research Practicum
Lecturer:	Abid Yaqoob
Submission Due Date:	11/08/2025
Project Title:	Enhancing Dublin Rental Price Prediction by Integrating Policy Insights via Large Language Models and Explainable AI
Word Count:	1272 **Page Count:** 13

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.
ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature: Emmanuella Anita Odiaka

Date: 06/08/2025

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST

Attach a completed copy of this sheet to each project (including multiple copies)	□
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	□
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	□

Assignments that are submitted to the Programme Coordinator Office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Configuration Manual

Emmanuella Anita Odiaka
Student ID: x23282819

Introduction

This report gives a detailed breakdown to the configuration and Implementation of the project.

1 Environment

This section details the environmental and system requirement setup that was used in implementing the project.

A Systems Configuration

Table 1: Software Configuration

Requirement	Specification
Programing Language	Python Version 3
Tools	Google Collab, WORD Open AI LLM
Google Drive	Access to Drive for the Data
Operating System	macOS Sequoia 15.6

Table 2: Hardware Configuration

Requirement	Specification
Processor	2.3 GHz 8-Core Intel Core i9
Memory	32 GB 2400 MHz DDR4
Computational resources	Google Colab's CPU, 12.7 GB RAM

B Dataset

The rental data was taken from Kaggle as seen in Figure 1. It originally had 2,718 rows of daft.ie rental data. The data was downloaded in CSV format and then uploaded to Google drive.

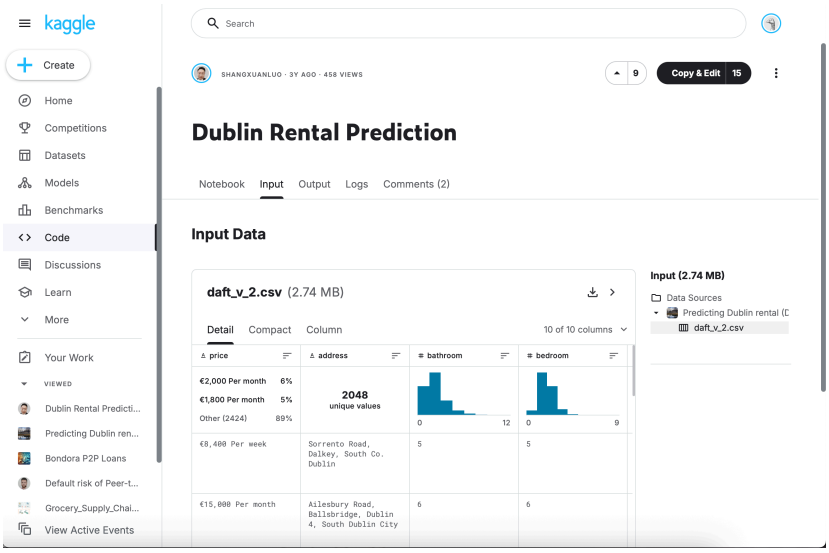


Figure 1: Rental Data Source

On the other hand, 14 government policy documents were coined from the internet using keyword searches like “Irish Rental Policy”. The report appendix shows the details of the documents used and the sources they were derived from.

C Google Colab Setup

The gathered data are stored in a Google Drive folder and accessed via Colab’s mount function as shown in Figures 2 and 3 respectfully.

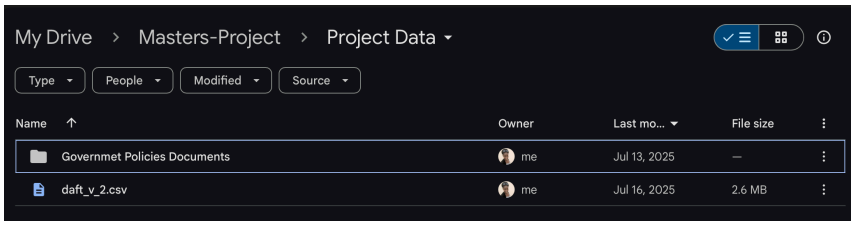


Figure 2: Data Location in Google Drive

The Mount function was used to build a connection between google drive and Colab to access the files as seen in Figure 3.

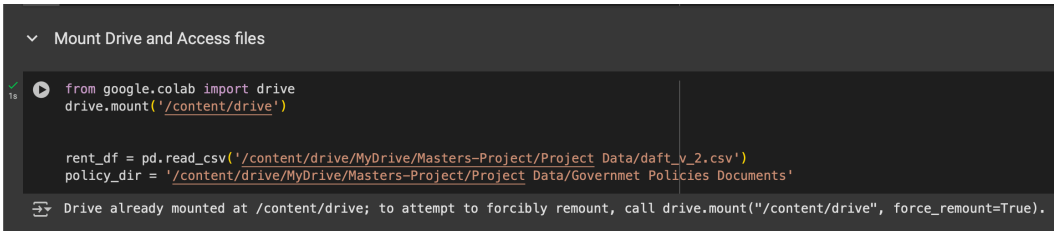


Figure 3: Loaded data using "Mount" from Colab

Figure 4 shows the selected language and processor engine used for the course of this project.

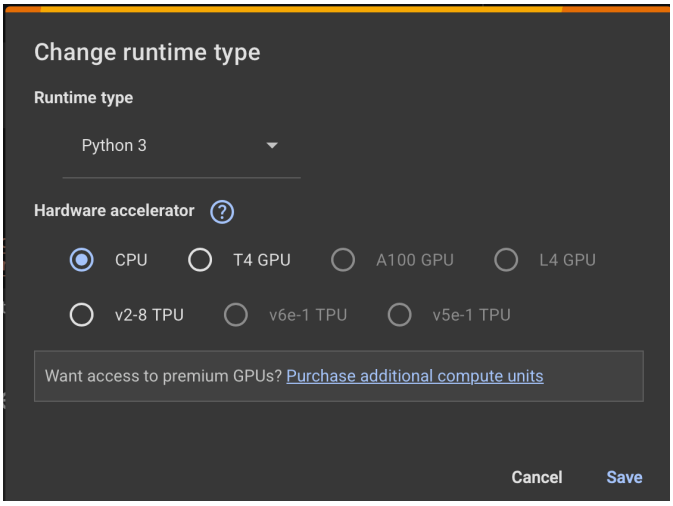


Figure 4: Colab Processor Engine

D Importing and Libraries

Figure 5 show the libraries imported and used for the implementation of the project.

```
import pandas as pd
import os
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import warnings # Ignore useless warnings (see SciPy issue #5998)
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score

warnings.filterwarnings(action="ignore", message="^internal gelsd")
```

```
import PyPDF2
import openai
import json
from getpass import getpass
import shap
import lime
import lime.lime_tabular
```

Figure 5: Libraries

2 Analysis and Evaluation of the Baseline model

This section will show the breakdown of the steps carried out to fully prepare and Analyse the baseline model.

A. Data Exploration and understanding

The process began by reading the basic information of the rental data and checking for null values and the variable types. This is seen in Figure 6.

```python
# Display the first 5 rows
print("First 5 rows of the DataFrame:")
display(rent_df.head())

# Get column names and their data types
print("\nColumn names and their data types:")
rent_df.info()

# Get a summary of numerical columns
print("\nSummary of numerical columns:")
display(rent_df.describe())

# Check for missing values
print("\nMissing values in each column:")
display(rent_df.isnull().sum())
```

```
Column names and their data types:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2718 entries, 0 to 2717
Data columns (total 10 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   price          2718 non-null   object
 1   address        2718 non-null   object
 2   bathroom       2718 non-null   int64
 3   bedroom        2718 non-null   int64
 4   furnish        2706 non-null   object
 5   description    2134 non-null   object
 6   property type  2718 non-null   object
 7   ID             2718 non-null   int64
 8   longitude      2718 non-null   float64
 9   latitude       2718 non-null   float64
dtypes: float64(2), int64(3), object(5)
memory usage: 212.5+ KB
```

Missing values in each column:

	0
price	0
address	0
bathroom	0
bedroom	0
furnish	12
description	584
property type	0
ID	0
longitude	0
latitude	0

dtype: int64

Figure 6: Basic information of rental Data

B. Data Preparation

For this process, three tasks outlined below were performed after which the cleaned data was sent to the model building phase.

- **Data cleaning**

  In this step, the missing values identified earlier were removed, price was converted to monthly values for consistency and the output was stored in a data frame for the Exploratory Data Analysis phase. This is seen in Figure 7.

```python
def comprehensive_data_cleaning(df):
    # 1. Create a copy of the dataframe
    df_copy = df.copy()

    # 1. Drop the 'description' column
    df_copy.drop('description', axis=1, inplace=True)


    # 2. Replaces null-like values with NaN (assuming empty strings or whitespace as null-like)
    df_copy.replace(r'^\s*$', np.nan, regex=True, inplace=True)

    # 3. Drop completely empty rows
    df_copy.dropna(how='all', inplace=True)

    # 4. Handle missing values in 'furnish'
    df_copy['furnish'] = df_copy['furnish'].fillna("Furnished or unfurnished")

    # 5. Convert 'price' column to numerical format and standardize to monthly
    def clean_price(price):
        if isinstance(price, str):
            price = price.replace('€', '').replace(',', '').strip()
            parts = price.split(' Per ')
            if len(parts) == 2:
                try:
                    value = float(parts[0])
                    period = parts[1].lower()
                    if 'week' in period:
                        return value * 4  # Approximate monthly price
                    elif 'month' in period:
                        return value
                    elif 'year' in period:
                        return value / 12 # Approximate monthly price
                except ValueError:
                    return np.nan
        return np.nan # Return NaN for unexpected formats or non-strings

    df_copy.loc[:, 'price'] = df_copy['price'].apply(clean_price)
    # Drop rows where price conversion failed
    df_copy.dropna(subset=['price'], inplace=True)

    # 6. Store the cleaned data in cleaned_data df
    cleaned_data = df_copy.copy()

    return cleaned_data

# Apply the cleaning function to your dataframe
cleaned_df = comprehensive_data_cleaning(rent_df)

# Display info and head of the cleaned dataframe to verify
print(f"Final cleaned data shape: {cleaned_df.shape}")

print("\nMissing values after cleaning:")
print(cleaned_df.isnull().sum())
```

```
Final cleaned data shape: (2718, 9)

Missing values after cleaning:
price          0
address        0
bathroom       0
bedroom        0
furnish        0
property type  0
ID             0
longitude      0
latitude       0
dtype: int64
```

Figure 7: Data Cleaning Process

- **Exploratory Data Analysis**

His process involved performing a correlation analysis and viewing the relationship between the variables. A snipet of the full process is as seen in figure 8.

Figure 8: EDA Process and Results

- **Outlier Handling**

Here, outliers were detected and handled with the InterQuartile Range method, using the created 'price_per_bedroom' feature. This process is highlighted in Figure 9.

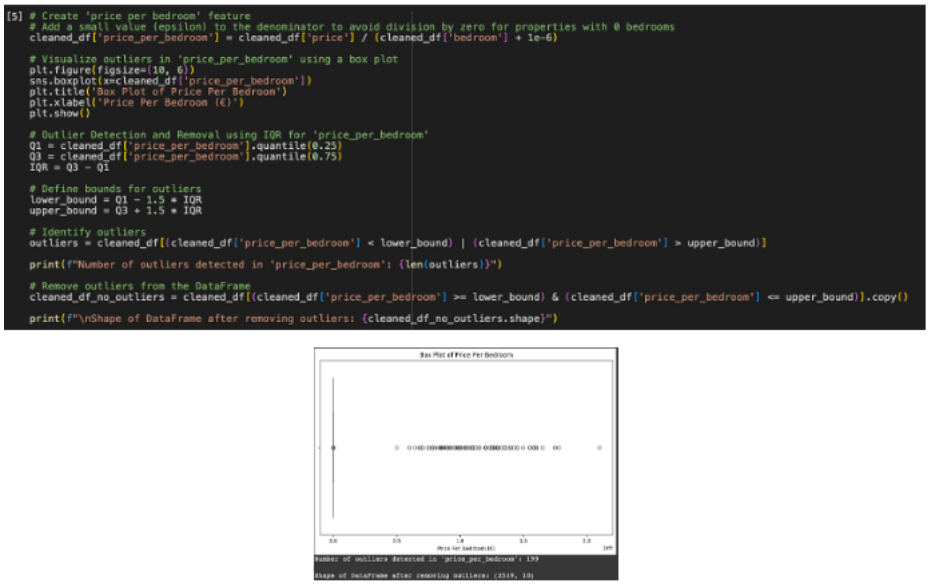
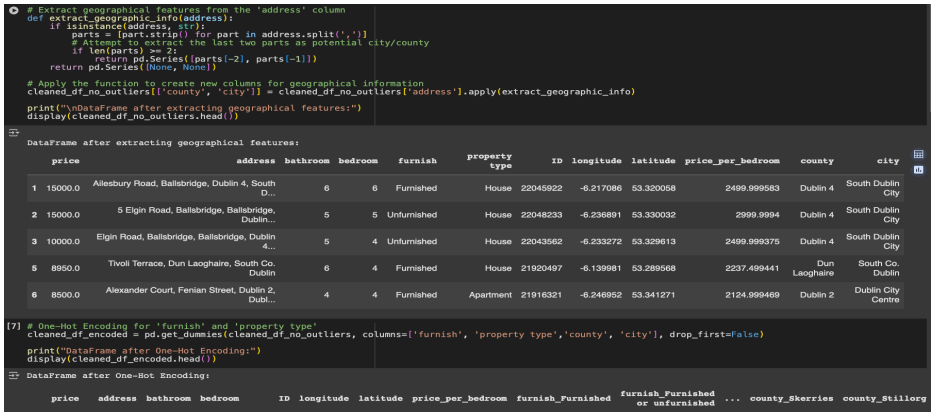


Figure 9: Outlier Handling Process and Result

- **Feature Engineering and Onehot Encoding**

Before sending the cleaned data for model building, the categorical variables were encoded and geographical features used in later analysis was engineered as seen in Figure 8.



```
# Extract geographical features from the 'address' column
def extract_geographic_info(address):
    if isinstance(address, str):
        parts = [part.strip() for part in address.split(',')]
        # Attempt to extract the last two parts as potential city/county
        if len(parts) >= 2:
            return pd.Series([parts[-2], parts[-1]])
    return pd.Series([None, None])

# Apply the function to create new columns for geographical information
cleaned_df_no_outliers[['county', 'city']] = cleaned_df_no_outliers['address'].apply(extract_geographic_info)

print("\nDataFrame after extracting geographical features:")
display(cleaned_df_no_outliers.head())
```

DataFrame after extracting geographical features:

	price	address	bathroom	bedroom	furnish	property type	ID	longitude	latitude	price_per_bedroom	county	city
1	15000.0	Ailesbury Road, Ballsbridge, Dublin 4, South D...	6	6	Furnished	House	22045922	-6.217086	53.320058	2499.999583	Dublin 4	South Dublin City
2	15000.0	5 Elgin Road, Ballsbridge, Ballsbridge, Dublin...	5	5	Unfurnished	House	22048233	-6.236891	53.330032	2999.9994	Dublin 4	South Dublin City
3	10000.0	Elgin Road, Ballsbridge, Ballsbridge, Dublin 4,...	5	4	Unfurnished	House	22043562	-6.233272	53.329613	2499.999375	Dublin 4	South Dublin City
5	8950.0	Tivoli Terrace, Dun Laoghaire, South Co. Dublin	6	4	Furnished	House	21920497	-6.139981	53.289568	2237.499441	Dun Laoghaire	South Co. Dublin
6	8500.0	Alexander Court, Fenian Street, Dublin 2, Dubl...	4	4	Furnished	Apartment	21916321	-6.246952	53.341271	2124.999469	Dublin 2	Dublin City Centre

```
[7] # One-Hot Encoding for 'furnish' and 'property type'
cleaned_df_encoded = pd.get_dummies(cleaned_df_no_outliers, columns=['furnish', 'property type','county', 'city'], drop_first=False)

print("DataFrame after One-Hot Encoding:")
display(cleaned_df_encoded.head())
```

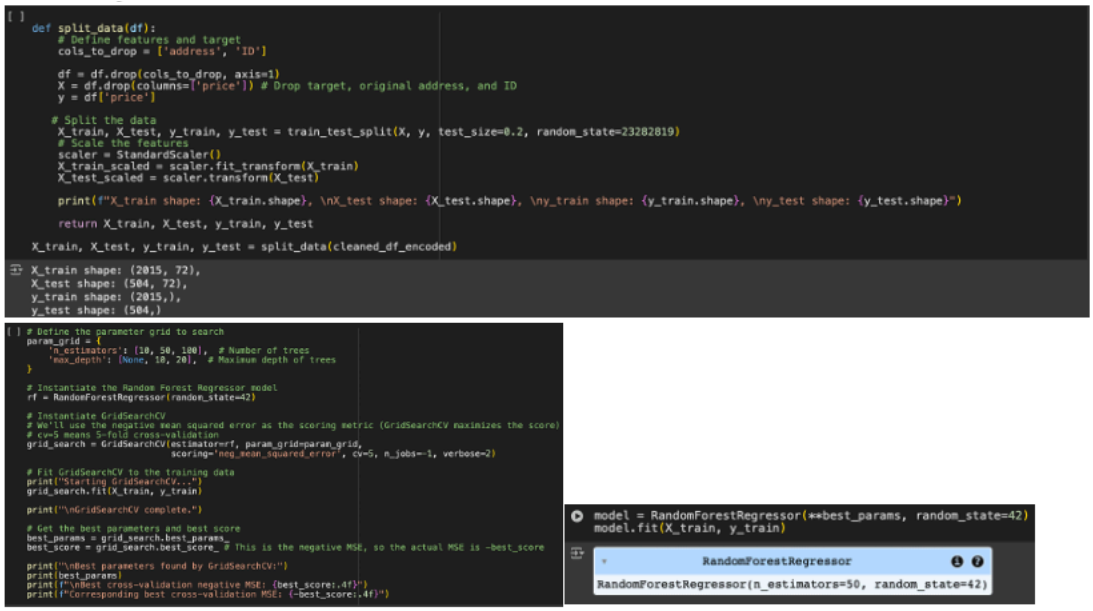
DataFrame after One-Hot Encoding:

	price	address	bathroom	bedroom	ID	longitude	latitude	price_per_bedroom	furnish_Furnished	furnish_Furnished or unfurnished	...	county_Skerries	county_Stillorg

Figure 10: Feature Engineering & One-Hot Encoding

C. Model Building

The next step involved splitting the data into training and test sets with an 80:20 ratio and then sending the training data into the RandomForestRegressor model for training. GridSearch CV was used to find the best parameters for the regression model. All this is evidenced in Figure 11 below.



```
def split_data(df):
    # Define features and target
    cols_to_drop = ['address', 'ID']

    df = df.drop(cols_to_drop, axis=1)
    X = df.drop(columns=['price']) # Drop target, original address, and ID
    y = df['price']

    # Split the data
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=23282819)
    # Scale the features
    scaler = StandardScaler()
    X_train_scaled = scaler.fit_transform(X_train)
    X_test_scaled = scaler.transform(X_test)

    print(f"X_train shape: {X_train.shape}, \nX_test shape: {X_test.shape}, \ny_train shape: {y_train.shape}, \ny_test shape: {y_test.shape}")

    return X_train, X_test, y_train, y_test

X_train, X_test, y_train, y_test = split_data(cleaned_df_encoded)
```

```
X_train shape: (2015, 72),
X_test shape: (504, 72),
y_train shape: (2015,),
y_test shape: (504,)
```

```
# Define the parameter grid to search
param_grid = {
    'n_estimators': [10, 50, 100],  # Number of trees
    'max_depth': [None, 10, 20],  # Maximum depth of trees
}

# Instantiate the Random Forest Regressor model
rf = RandomForestRegressor(random_state=42)

# Instantiate GridSearchCV
# We'll use the negative mean squared error as the scoring metric (GridSearchCV maximizes the score)
# cv=5 means 5-fold cross-validation
grid_search = GridSearchCV(estimator=rf, param_grid=param_grid,
                           scoring='neg_mean_squared_error', cv=5, n_jobs=-1, verbose=2)

# Fit GridSearchCV to the training data
print("Starting GridSearchCV...")
grid_search.fit(X_train, y_train)

print("\nGridSearchCV complete.")

# Get the best parameters and best score
best_params = grid_search.best_params_
best_score = grid_search.best_score_ # This is the negative MSE, so the actual MSE is -best_score

print("\nBest parameters found by GridSearchCV:")
print(best_params)
print(f"\nBest cross-validation negative MSE: {best_score:.4f}")
print(f"Corresponding best cross-validation MSE: {-best_score:.4f}")
```

```
model = RandomForestRegressor(**best_params, random_state=42)
model.fit(X_train, y_train)
```

```
RandomForestRegressor
RandomForestRegressor(n_estimators=50, random_state=42)
```

Figure 11: Baseline Modeling

D. Evaluation

Appropriate metrics were used to evaluate the performance of the model on the test data, after which feature importance analysis was carried out to see the top predictive features that most influenced the model, as shown in Figure 12.

```
[ ] # Make predictions on the test set
    y_pred = model.predict(X_test)

    # Calculate evaluation metrics
    mae = mean_absolute_error(y_test, y_pred)
    mse = mean_squared_error(y_test, y_pred)
    rmse = np.sqrt(mse)
    r2 = r2_score(y_test, y_pred)

    # Print the evaluation metrics
    print(f"Mean Absolute Error (MAE): {mae:.2f}")
    print(f"Mean Squared Error (MSE): {mse:.2f}")
    print(f"Root Mean Squared Error (RMSE): {rmse:.2f}")
    print(f"R-squared (R2): {r2:.2f}")
```

```
# Get feature importances from the trained model
feature_importances = model.feature_importances_

if 'cleaned_df_encoded' in locals():
    df_for_features = cleaned_df_encoded.copy()
    target_column = 'price'
    X_for_features = df_for_features.drop(columns=[target_column, 'address', 'ID'])

    feature_importances_series = pd.Series(feature_importances, index=X_train.columns)

    # Get the top 10 most important features
    top_10_features = feature_importances_series.nlargest(10)

    # Visualize the feature importances
    top10_features_sorted = top_10_features.sort_values(ascending=False)

    plt.figure(figsize=(10, 7))
    sns.barplot(x=top10_features_sorted.values, y=top10_features_sorted.index, palette='viridis')
    plt.title('Top 10 Most Important Features (Baseline Model)')
    plt.xlabel('Importance')
    plt.ylabel('Feature')
    plt.show()

    print("Top 10 Most Important Features:")
    print(top_10_features)

else:
    print("Error: cleaned_df_encoded DataFrame not found. Cannot determine feature names.")
```

Figure 12: Baseline Model Evaluation & Feature Importance

3 Prerequisites for the Policy Documents Processing

This section focuses on the steps taken to prepare the policy documents for the data extraction process.

A. Data preparation

Installs and file loading: First, the libraries needed for this section were installed and imported into the notebook as has been shown in Section 2D. After that, the policy files were loaded, and 2 samples were read to examine the structure and identify potential challenges that could be encountered in the extraction step. This is shown in Figure 13.

```
Load and explore the structure and content of the PDF Files

policy_files = os.listdir(policy_dir)
print("Files in the policy documents directory:")
for file in policy_files:
    print(file)

Files in the policy documents directory:
Planning and Development (Housing) and Residential Tenancies Act 2016.pdf
Residential Policy Covid.pdf
Affordable Housing Act 2021.pdf
Housing Policy: Actions to Deliver Change.pdf
Proposed Changes to Irish Residential Rent Controls.pdf
Revised Residential Tenancies and Valuation Act 2020.pdf
Residential Tenancies Bill 2021.pdf
Residential Tenancies (No. 2) Act 2021.pdf
Residential Tenancies Act 2021.pdf
RESIDENTIAL TENANCIES ACT 2004 REVISED Updated to 15 May 2025.pdf
housing-for-all-a-new-housing-plan-for-ireland-executive-summary.pdf
Cost rental housing.pdf
The Impact of Cost Rental Housing-5.pdf
Final Review RPZ and Policy Options for Rent Controls in PRS.pdf
Housing Assistance Payment (HAP).pdf
```

```
[ ]
# Select a couple of files for exploration
selected_files = [
    'Planning and Development (Housing) and Residential Tenancies Act 2016.pdf',
    'Affordable Housing Act 2021.pdf'
]

print("\nExploring content of selected PDF files:")
for file_name in selected_files:
    file_path = os.path.join(policy_dir, file_name)
    print(f"\n--- Content of {file_name} ---")
    try:
        with open(file_path, 'rb') as f:
            reader = PyPDF2.PdfReader(f)
            num_pages = len(reader.pages)
            print(f"Number of pages: {num_pages}")
            # Read content from the first few pages to get an idea of the structure
            content = ""
            for i in range(min(num_pages, 3)): # Read first 3 pages
                page = reader.pages[i]
                content += page.extract_text() or "" # Use empty string for None

            # Print a truncated version of the content
            print(content[:1000] + "...") # Print first 1000 characters

    except FileNotFoundError:
        print(f"Error: File not found at {file_path}")
    except Exception as e:
        print(f"An error occurred while reading {file_name}: {e}")
```

Figure 13: Policy Documents Loading and Reading

OpenAI API Key Input: The API key used to access Open AI was prompted and stored as an environment variable and used to query the API as seen in Figure 14.

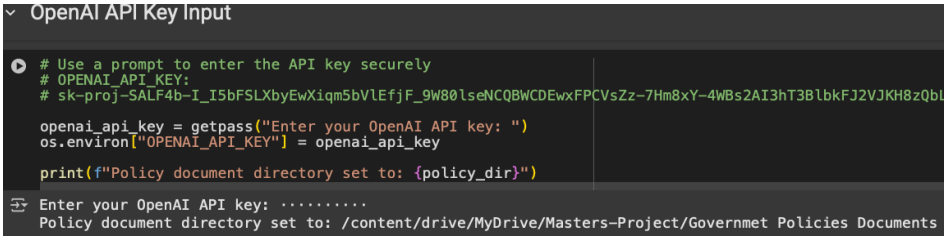


Figure 14: LLM API Accessibility

B. Extraction Logic

- **Initial Extraction Logic:** The process entailed defining a prompt template for the LLM and deriving the extracted information in chunks of 10000-character limit to send to the API and storing the resulting JSON output in a list that was later converted to a Pandas Dataframe. A snippet of this process is shown in Figure 15.

Figure 15: Initial LLM Extraction Logic

The next step was to engineer features focused on policies related to Dublin, so the next step in the process was to study the information contained within the geographical_areas feature of the new dataset. This is shown in Figure 16.

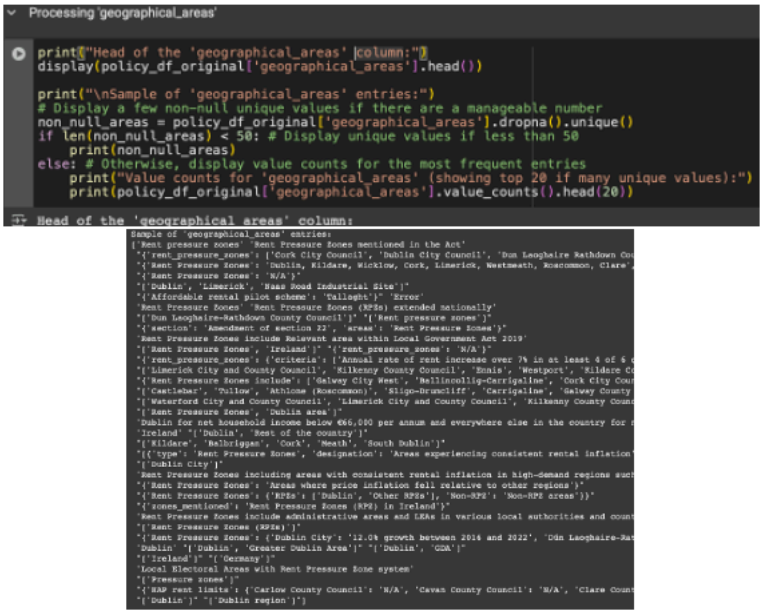


Figure 16: Listing of Geographical Areas from Extraction

- **Refined Extraction Logic:** Based on how complex and varied the resulting data attributes from the data was, a new stratergy with a refined logic was implemented to drive the extraction to focus specifically on Dublin area – a snipet of which is seen in Figure 17.

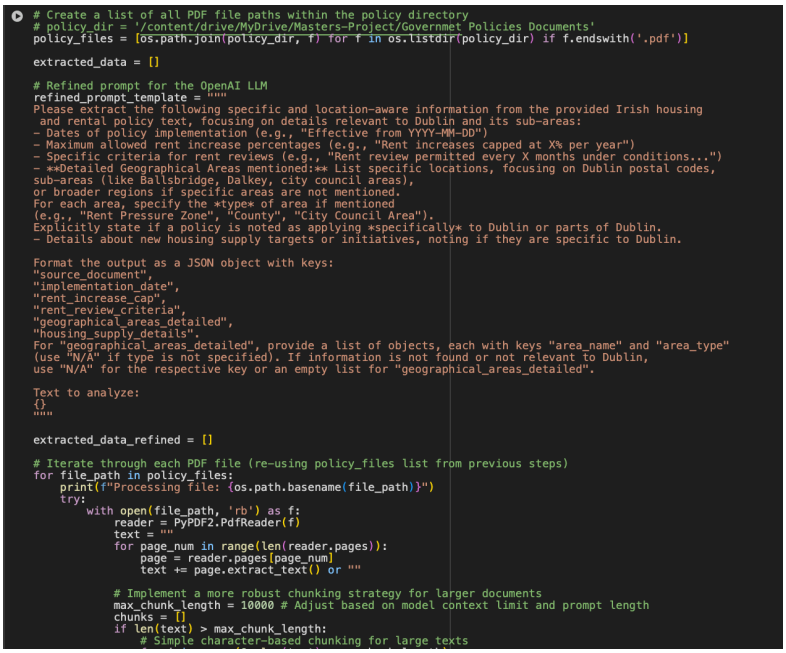


Figure 17: Refined LLM Extraction Logic

C. Hybrid Dataset creation

The steps followed to generate the hybrid dataset started from identifying and aggregating relevant key words or features from the extracted data and then linking them to individual rental properties data to generate the hybrid dataset. These steps are shown in Figures 18 – 20

Aggregate policy features creation is on the presence of specific keywords like rent increase, review criteria and housing supply details as seen in figure 18. Here, the merging of the original cleaned baseline dataset and the aggregated features was carried out.

```python
# Let's create features based on the presence of specific keywords or non-'N/A' values
# We'll count how many entries in policy_df (which came from chunks of documents)

# Count non-'N/A' entries in relevant columns as a proxy for policy mentions
num_rent_increase_mentions = policy_df[policy_df['rent_increase_cap'] != 'N/A'].shape[0]
num_rent_review_mentions = policy_df[policy_df['rent_review_criteria'] != 'N/A'].shape[0]
num_housing_supply_mentions = policy_df[policy_df['housing_supply_details'] != 'N/A'].shape[0]

print(f"Number of policy entries mentioning rent increase caps: {num_rent_increase_mentions}")
print(f"Number of policy entries mentioning rent review criteria: {num_rent_review_mentions}")
print(f"Number of policy entries mentioning housing supply details: {num_housing_supply_mentions}")

# Create new columns in the original DataFrame with these aggregate counts.
df_hybrid = cleaned_df_encoded.copy() # Create a copy to avoid modifying the original df directly

df_hybrid['policy_rent_increase_mentions'] = num_rent_increase_mentions
df_hybrid['policy_rent_review_mentions'] = num_rent_review_mentions
df_hybrid['policy_housing_supply_mentions'] = num_housing_supply_mentions

print("\nDataFrame with added aggregate policy features:")
display(df_hybrid.head())
```

```
Number of policy entries mentioning rent increase caps: 180
Number of policy entries mentioning rent review criteria: 180
Number of policy entries mentioning housing supply details: 180

DataFrame with added aggregate policy features:
```

Figure 18: Aggregated Hybrid Dataset Creation

Geographical areas features were standardised to normal location attributes like “Dublin”, “Lucan”, “Killiney”, etc as seen in Figure 16, which was then used to filter out Dublin specific policies from the refined dataframe as seen in Figure 19.

```python
policy_df_refined = policy_df.copy()
def extract_and_standardize_locations(area_list):
    locations = []
    if isinstance(area_list, list):
        for area_info in area_list:
            if isinstance(area_info, dict) and 'area_name' in area_info:
                area_name = area_info['area_name']
                # Simple standardization: convert to lower case and remove leading/trailing whitespace
                standardized_name = area_name.lower().strip()
                # Further standardization involve mapping variations (e.g., "Dublin City Council" to "Dublin")
                locations.append(standardized_name)
    return list(set(locations)) # Return unique standardized locations

policy_df_refined['standardized_locations'] = policy_df_refined['geographical_areas_detailed'].apply(extract_and_standardize_locations)

# Examine the extracted standardized locations
print("\nSample of standardized locations extracted from policy documents:")
display(policy_df_refined['standardized_locations'].head())
```

```python
# Get unique values from the 'county' column
if 'county' in cleaned_df_no_outliers.columns:
    unique_counties = cleaned_df_no_outliers['county'].unique()
    unique_counties = [str(item).lower() for item in unique_counties if isinstance(item, str)]
    print("Unique values in the 'county' column of cleaned_df_no_outliers:")
    print(unique_counties)
else:
    print("Error: 'county' column not found in cleaned_df_no_outliers.")

print("-" * 30)

# Get unique values from the 'city' column
if 'city' in cleaned_df_no_outliers.columns:
    unique_cities = cleaned_df_no_outliers['city'].unique()
    unique_cities = [str(item).lower() for item in unique_cities if isinstance(item, str)]
    print("Unique values in the 'city' column of cleaned_df_no_outliers:")
    print(unique_cities)
else:
    print("Error: 'city' column not found in cleaned_df_no_outliers.")
```

```python
# Given the variability, a simple approach is to create features indicating
# the presence of any Dublin-related location mention in a policy entry.

def is_dublin_related(locations):
    if not locations:
        return False
    # Check if 'dublin' or any Dublin-specific keywords are in the standardized locations
    dublin_keywords = [] # Add other general keywords as needed

    # Add unique Dublin counties and cities from your rental data (in lowercase)
    # Ensure you only add relevant Dublin-area counties/cities if your unique lists contain non-Dublin areas
    dublin_keywords.extend(unique_counties)
    dublin_keywords.extend(unique_cities)

    # Convert to a set and back to a list to remove duplicates and keep it clean
    dublin_keywords = sorted(list(set(dublin_keywords)))

    # print(f"Generated Dublin keywords for filtering: {dublin_keywords}")
    for loc in locations:
        if any(keyword in loc for keyword in dublin_keywords):
            return True
        # Also check if area_type indicates a Dublin-related zone if available
        # (This would require modifying the extract_and_standardize_locations to return type as well)
        # For now, rely on name matching.
    return False

policy_df_refined['is_dublin_related_policy'] = policy_df_refined['standardized_locations'].apply(is_dublin_related)
```

```python
# Count the number of policy entries (chunks) that are considered Dublin-related
num_dublin_related_policy_mentions = policy_df_refined['is_dublin_related_policy'].sum()

# Count mentions of specific policy types relevant to Dublin (e.g., RPZ)
# This requires examining the original extracted data before standardization if type info is needed
# Let's re-process the raw geographical_areas_detailed to look for RPZ type mentions
def count_rpz_mentions(area_list):
    count = 0
    if isinstance(area_list, list):
        for area_info in area_list:
            if isinstance(area_info, dict) and 'area_type' in area_info and 'Rent Pressure Zone' in area_info['area_type']:
                count += 1
            elif isinstance(area_info, dict) and 'area_name' in area_info and 'Rent Pressure Zone' in area_info['area_name']: # Sometimes type is in name
                count += 1
    return count

policy_df_refined['rpz_mentions_count'] = policy_df_refined['geographical_areas_detailed'].apply(count_rpz_mentions)

total_rpz_mentions_in_dublin_related_policies = policy_df_refined[policy_df_refined['is_dublin_related_policy']]['rpz_mentions_count'].sum()


print(f"\nNumber of policy entries identified as Dublin-related: {num_dublin_related_policy_mentions}")
print(f"Total count of RPZ mentions within Dublin-related policy entries: {total_rpz_mentions_in_dublin_related_policies}")
```

Figure 19: Standardization and Filtering of Refined Dataset

Finally, these features were merged with the aggregated policy features and this formed the newly defined Hybrid Dataset as shown in Figure 20.

```
[ ] # Count the number of policy entries (chunks) that are considered Dublin-related
    num_dublin_related_policy_mentions = policy_df_refined['is_dublin_related_policy'].sum()

    # Count mentions of specific policy types relevant to Dublin (e.g., RPZ)
    # This requires examining the original extracted data before standardization if type info is needed
    # Let's re-process the raw geographical_areas_detailed to look for RPZ type mentions
    def count_rpz_mentions(area_list):
        count = 0
        if isinstance(area_list, list):
            for area_info in area_list:
                if isinstance(area_info, dict) and 'area_type' in area_info and 'Rent Pressure Zone' in area_info['area_type']:
                    count += 1
                elif isinstance(area_info, dict) and 'area_name' in area_info and 'Rent Pressure Zone' in area_info['area_name']: # Sometimes type is in name
                    count += 1
        return count

    policy_df_refined['rpz_mentions_count'] = policy_df_refined['geographical_areas_detailed'].apply(count_rpz_mentions)

    total_rpz_mentions_in_dublin_related_policies = policy_df_refined[policy_df_refined['is_dublin_related_policy']]['rpz_mentions_count'].sum()


    print(f"\nNumber of policy entries identified as Dublin-related: {num_dublin_related_policy_mentions}")
    print(f"Total count of RPZ mentions within Dublin-related policy entries: {total_rpz_mentions_in_dublin_related_policies}")
```

Figure 20: Final Hybrid Dataset

4 Policy Driven Implementation

A. Data Splitting

Following the derivation of the Hybrid dataset, the following steps include splitting of the Hybrid dataset into training and testing subsets using the split function earlier defined as shown in Figure 21

```
X_hybrid_refined, X_hybrid_refined_test, y_hybrid_refined, y_hybrid_refined_test = split_data(df_hybrid_refined)

print("\nPrepared Refined Hybrid Feature set (X_hybrid_refined) head:")
display(X_hybrid_refined.head())
print("\nPrepared Refined Hybrid Target variable (y_hybrid_refined) head:")
display(y_hybrid_refined.head())
print("\nInfo of the Prepared Refined Hybrid Feature set (X_hybrid_refined):")
X_hybrid_refined.info()
```

```
X_train shape: (2015, 77),
X_test shape: (504, 77),
y_train shape: (2015,),
y_test shape: (504,)

Prepared Refined Hybrid Feature set (X_hybrid_refined) head:

   bathroom  bedroom  longitude  latitude  price_per_bedroom  furnish_Furnished  furnish_Furnished or unfurnished  furnish_Unfurni
```

Figure 21: Split Hybrid Data

B. Model Building and Training

The earlier defined regressor model along with the same parameters are also used to test the hybrid model as shown in Figure 22.

```
hybrid_model = RandomForestRegressor(**best_params, random_state=42)
hybrid_model.fit(X_hybrid_refined, y_hybrid_refined)
```

```
RandomForestRegressor
RandomForestRegressor(n_estimators=50, random_state=42)
```

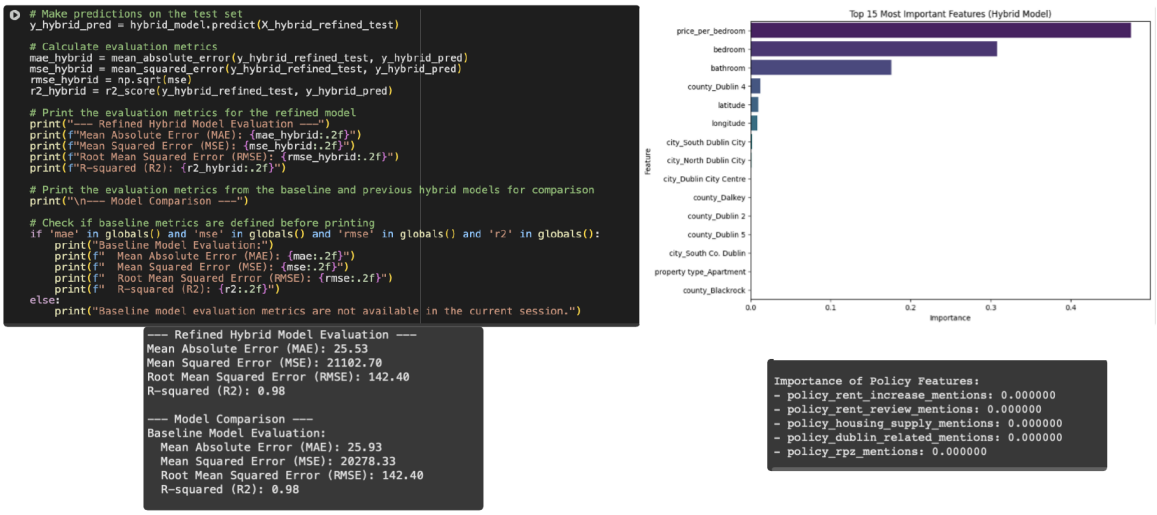
Figure 22: Hybrid Model Training

5 Evaluation and Explainability

This section covers the evaluation performed on the dataset, and the tools used for the model's explainability.

D. Model Evaluation

The same feature importance analysis and metrics used to evaluate the baseline model, was used in this step as seen in Figure 23, along with a comparison with the baseline model's performance.




```
# Make predictions on the test set
y_hybrid_pred = hybrid_model.predict(X_hybrid_refined_test)

# Calculate evaluation metrics
mae_hybrid = mean_absolute_error(y_hybrid_refined_test, y_hybrid_pred)
mse_hybrid = mean_squared_error(y_hybrid_refined_test, y_hybrid_pred)
rmse_hybrid = np.sqrt(mse)
r2_hybrid = r2_score(y_hybrid_refined_test, y_hybrid_pred)

# Print the evaluation metrics for the refined model
print("--- Refined Hybrid Model Evaluation ---")
print(f"Mean Absolute Error (MAE): {mae_hybrid:.2f}")
print(f"Mean Squared Error (MSE): {mse_hybrid:.2f}")
print(f"Root Mean Squared Error (RMSE): {rmse_hybrid:.2f}")
print(f"R-squared (R2): {r2_hybrid:.2f}")

# Print the evaluation metrics from the baseline and previous hybrid models for comparison
print("\n--- Model Comparison ---")

# Check if baseline metrics are defined before printing
if 'mae' in globals() and 'mse' in globals() and 'rmse' in globals() and 'r2' in globals():
    print("Baseline Model Evaluation:")
    print(f"  Mean Absolute Error (MAE): {mae:.2f}")
    print(f"  Mean Squared Error (MSE): {mse:.2f}")
    print(f"  Root Mean Squared Error (RMSE): {rmse:.2f}")
    print(f"  R-squared (R2): {r2:.2f}")
else:
    print("Baseline model evaluation metrics are not available in the current session.")
```

```
--- Refined Hybrid Model Evaluation ---
Mean Absolute Error (MAE): 25.53
Mean Squared Error (MSE): 21102.70
Root Mean Squared Error (RMSE): 142.40
R-squared (R2): 0.98

--- Model Comparison ---
Baseline Model Evaluation:
  Mean Absolute Error (MAE): 25.93
  Mean Squared Error (MSE): 20278.33
  Root Mean Squared Error (RMSE): 142.40
  R-squared (R2): 0.98
```

```
Importance of Policy Features:
- policy_rent_increase_mentions: 0.000000
- policy_rent_review_mentions: 0.000000
- policy_housing_supply_mentions: 0.000000
- policy_dublin_related_mentions: 0.000000
- policy_rpz_mentions: 0.000000
```

Figure 23: Hybrid Model Evaluation and Feature Importance

E. Model Explainability

The hybrid model was further evaluated using SHAP and LIME to explain the model's predictions. The implementation and results are as seen in figures 24 and 25 respectively.

```
# Create an explainer object using the RandomForestRegressor tree explainer.
explainer = shap.TreeExplainer(hybrid_model)

# Calculate SHAP values for the test set
print("Calculating SHAP values...")
shap_values = explainer.shap_values(X_hybrid_refined_test)
print("SHAP values calculation complete.")
```

```
Calculating SHAP values...
SHAP values calculation complete.
```

```
# Visualize the SHAP summary plot
print("Generating SHAP summary plot...")
shap.summary_plot(shap_values, X_hybrid_refined_test, plot_type="bar", show=False)
plt.title("SHAP Feature Importance (Hybrid Model)")
plt.show()

shap.summary_plot(shap_values, X_hybrid_refined_test, show=False)
plt.title("SHAP Summary Plot (Hybrid Model)")
plt.show()


# Visualize dependence plots for few important features based on the summary plot
important_features = ['price_per_bedroom', 'bedroom', 'bathroom', ''] # Add other features of interest

print("\nGenerating SHAP dependence plots for important features...")
for feature in important_features:
    if feature in X_hybrid_refined_test.columns:
        shap.dependence_plot(feature, shap_values, X_hybrid_refined_test, interaction_index=None, show=False)
        plt.title(f"SHAP Dependence Plot for {feature}")
        plt.show()
    else:
        print(f"Feature '{feature}' not found in the test set.")
```

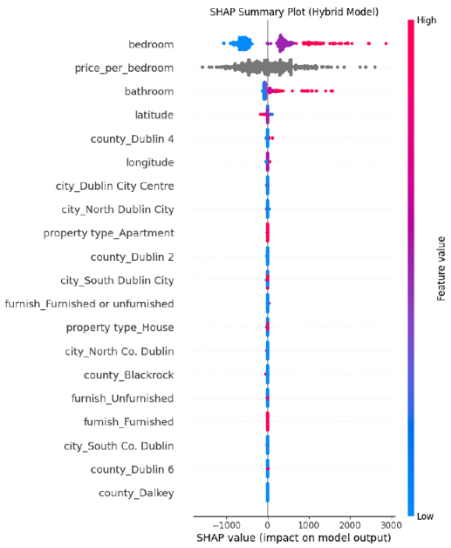


Figure 24: SHAP Implementation and Results

```
# Get the representative sample from the hybrid test data
feature_names = X_hybrid_refined_test.columns.tolist()

# Create a LIME explainer
explainer_lime = lime.lime_tabular.LimeTabularExplainer(
    training_data=X_hybrid_refined.values, # Use the training data values (numpy array)
    feature_names=feature_names,
    mode="regression"
)

# Choose an instance from the test set to explain (e.g., the first instance)
instance_idx = 0
instance_to_explain = X_hybrid_refined_test.iloc[instance_idx].values # Get the values as a numpy array
true_price = y_hybrid_refined_test.iloc[instance_idx] # Get the true price
predicted_price = hybrid_model.predict(instance_to_explain.reshape(1, -1))[0] # Predict price for the instance

print(f"Explaining prediction for instance {instance_idx}:")
print(f"True Price: {true_price:.2f}")
print(f"Predicted Price: {predicted_price:.2f}")

# Generate an explanation for the chosen instance
# num_features: number of features to include in the explanation
explanation = explainer_lime.explain_instance(
    data_row=instance_to_explain,
    predict_fn=hybrid_model.predict,
    num_features=10 # You can adjust the number of features
)

# Display the explanation
print("\nLIME Explanation:")
explanation.as_list()
```

```
[ ] # Visualize the LIME explanation
    # This will typically open in a new browser tab or render in the notebook output
    print("\nVisualizing LIME explanation:")
    explanation.show_in_notebook(show_table=True, show_all=False)
```

```
Visualizing LIME explanation:
```

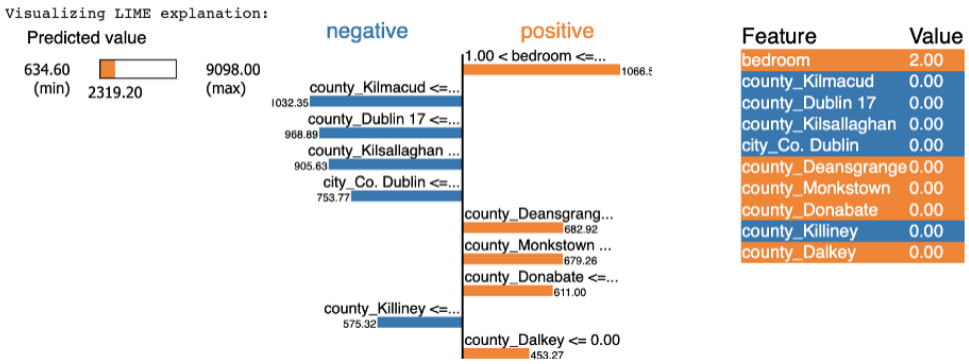


Figure 25: LIME Implementation and Results

6 Conclusion

This document presets a step-by-step approach in representing the technical implementation of this research. To replicate the code and achieve the same results and gain a deeper understanding for the inner working of the project, follow this outlined pathway.

Report Presentation video link
Code Presentation video link